2011

Enumeration of inequivalent cycle decompositions

William J. Laffin

Michigan Technological University

ENUMERATION OF INEQUIVALENT CYCLE DECOMPOSITIONS

By
WILLIAM J. LAFFIN

A REPORT
Submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
(Mathematical Sciences)

MICHIGAN TECHNOLOGICAL UNIVERSITY
2011

This report, "Enumeration of Inequivalent Cycle Decompositions," is hereby approved in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN MATHEMATICAL SCIENCES.

Department of Mathematical Sciences:

Signatures:

Thesis Advisor _____
Donald L. Kreher, Ph.D.

Committee Member _____
Phillip Merkey, Ph.D.

Committee Member _____
Charles Wallace, Ph.D.

Department Chair _____
Mark S. Gockenbach, Ph.D.

Date _____

Contents

Abstract

A k-cycle decomposition of order n is a partition of the edges of the complete graph on n vertices into k-cycles. In this report a backtracking algorithm is developed to count the number of inequivalent k-cycle decompositions of order n.

Chapter 1

Preliminary Theory and History

In this chapter we discuss the relevant background with a short survey of results on cycle enumeration.

1.1 Graphs and Decompositions

A *graph* X is a pair $(V(X), E(X))$, or simply (V, E) such that V is a set of points and E is a multi-set of unordered pairs of points from V. The set V is referred to as the vertex set while the set E is referred to as the edge set. A *simple graph* is one such that V is a finite set and E is a set without repetition. The *order* of a graph is given by $|V(X)|$ while the *size* of a graph is given by $|E(X)|$. The *degree* of a vertex $x \in V(X)$ is the size of the set $\{e \in E(X) : x \in e\}$. For example the graph Y in Figure 1.1 has

$$V(Y) = \{0, 1, 2, 3, 4, 5\}$$

and

$$E(Y) = \{\{0,3\}, \{0,4\}, \{3,4\}, \{1,3\}, \{1,5\}, \{3,5\}, \{2,4\}, \{2,5\}, \{4,5\}\}$$

The size of Y is 9, while the order of Y is 6.

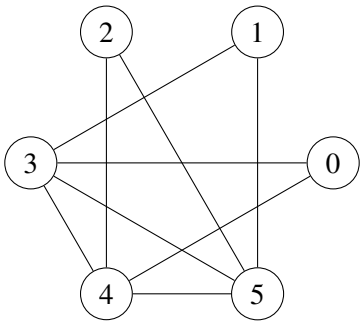
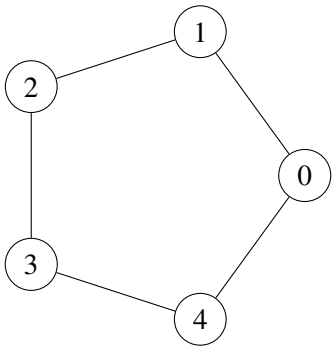


Figure 1.1: The graph Y



Flips:	(1,4)(2,3),	(2,0)(3,4),	(3,1)(4,0),	(4,2)(0,1),	(0,3)(1,2).
Rotations:	(0)(1)(2)(3)(4),	(0,1,2,3,4),	(0,2,4,1,3),	(0,3,1,4,2),	(0,4,3,2,1).

Figure 1.2: The automorphism group of C_5 is $D_{2\cdot 5}$

The graph of all possible edges K_n is called the *complete graph*. A *subgraph* Y of a graph X is a graph such that $V(Y) \subseteq V(X)$ and $E(Y) \subseteq E(X)$. For example, a subgraph y of Y in Figure 1.1 is given by

$$V(y) = \{3, 4, 5\}$$

and

$$E(y) = \{\{3, 4\}, \{4, 5\}, \{3, 5\}\}.$$

Unless stated otherwise, in this report all graphs are given the vertex set $\{0, 1, \ldots, n-1\}$. A *walk* on a graph X is an array $A = (a_0, a_1, \ldots, a_{l-1})$ of vertices such that $\{a_i, a_{i+1}\} \in E(X)$. A *path* on a graph X is a walk such that no two vertices are the same. A *cycle* is a subgraph C_k of a graph X such that the vertices $V(C_k)$ can be arranged $c_0, c_1, \ldots, c_{k-1}$ such that

$$E(C_k) = \{\{c_i, c_{i+1 \mod k}\} : 0 \leq i < k\}.$$

We define k the length of a cycle to be the number of vertices. For example, C_4 can be seen in graph Y in Figure 1.1 by the array $1, 3, 4, 5$.

An *automorphism* ρ of a graph X is a bijection on $V(X)$ such that

$$\{\{\rho(x), \rho(y)\} : \{x, y\} \in E(X)\} = E(X)$$

The permutation $1 \to 1, 4 \to 4, 3 \to 5, 5 \to 3, 0 \to 2, 2 \to 0$ is an automorphism of Y from Figure 1.1. Two graphs X_1 and X_2 are *isomorphic* if there is a bijection $\rho : V(X_1) \to V(X_2)$ such that

$$\{\rho(x) : x \in V(X_1)\} = V(X_2)$$
$$\{\{\rho(x), \rho(y)\} : \{x, y\} \in E(X_1)\} = E(X_2)$$

The automorphism group of a graph is the subgroup $\mathrm{Aut}(X)$ of S_V that consists of all the automorphisms of X. For example, the *dihedral group* D_{2n} is the group of all automorphisms of the cycle on n vertices. The size of this group is $2n$ and consists of "flips" and "rotations". In Figure 1.2 $D_{2\cdot 5}$ is shown in permutation notation, where (a, b, c) means $a \to b \to c \to a$.

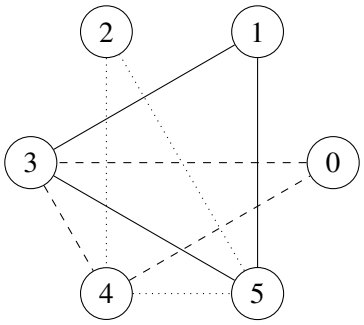


Figure 1.3: A 3-cycle decomposition of the graph Y in Figure 1.1.

A *decomposition* of a graph X into isomorphic copies of a subgraph Y is a partition $\mathscr{D} = \{d_1, \ldots, d_k\}$ of $E(X)$ into disjoint sets such that each graph defined by $(\bigcup d_i, d_i)$ is isomorphic to Y. $\bigcup d_i$ here refers to the union of all edges in d_i. A *k-cycle decomposition* of a graph X is an edge decomposition of X into k-cycles. A 3-cycle decomposition of the graph Y from Figure 1.1 is provided in Figure 1.3. Two k-cycle decompositions $\mathscr{D}_1, \mathscr{D}_2$ of a graph X are said to be *isomorphic* if there is a $\rho \in \text{Aut}(X)$ such that $d_i \in \mathscr{D}_1$ if and only if $\rho(d_i) \in \mathscr{D}_2$. An *automorphism* of a k-cycle decomposition is an isomorphism between a decomposition and itself. A *cyclic* k-cycle decomposition is one which $V(X) = \mathbb{Z}_n$ and $\rho(v) = v + 1 \mod n$ is an automorphism. We specify a k-cycle decomposition as a sequence of vertices partitioned into k-cycles. For example the 3-cycle decomposition of the graph Y displayed in Figure 1.3 can be specified as

$$A = (2,4,5)(1,3,5)(3,4,0)$$

or

$$A = (0,3,4)(1,3,5)(2,4,5)$$

Two k-cycle decompositions are *equivalent* if they consist of the same set of cycles, otherwise they are *inequivalent*. (In Chapter 2, for ease of exposition we often drop the inner parentheses and just write the sequence of vertices $A = (a_0, a_1, \ldots, a_{m-1})$).

The *k-cycle decomposition problem* is finding k-cycle decompositions. Cycle Decomposition problems in general are NP-complete [5, 11]. A recent survey on cycle decompositions is [3].

For the purpose of enumeration we define 3 quantities, all on vertex set $\{0, 1, \ldots, n-1\}$:

- $\mathscr{M}(n,k)$ is the number of inequivalent k-cycle decompositions of K_n.
- $\mathscr{N}(n,k)$ the number of pairwise non-isomorphic k-cycle decompositions of K_n.
- $\mathscr{C}(n,k)$ the number of pairwise non-isomorphic cyclic k-cycle decompositions of K_n.

Table 1.1 summarizes known values of $\mathscr{M}(n,k)$, $\mathscr{N}(n,k)$, and $\mathscr{C}(n,k)$.

Problem: What is the number of inequivalent k-cycle decompositions of the complete graph of K_n? What is $\mathscr{M}(n,k)$?.

Table 1.1: Enumeration Results.

n	k	$\mathscr{C}(n,k)$	ref	$\mathscr{N}(n,k)$	ref	$\mathscr{M}(n,k)$	ref
5	5	?	–	?	–	6	[10]
7	3	1	[1]	1	–	30	–
7	7	?	–	?	–	960	[10]
9	3	0	[7]	1	–	840	–
9	4	1	[1]	?	–	$1,643,040$	New
9	6	?	–	?	–	$222,243,840$	New
9	9	?	–	122	[4]	$40,037,760$	[10]
11	5	4	[1]	?	–	?	–
11	11	?	–	?	–	?	–
13	3	1	[7]	2	[6]	$1,197,504,000$	[6]
13	6	16	[1]	?	–	?	–
13	13	?	–	?	–	?	–
15	3	2	–	80	[6]	$60,281,712,691,200$	[6]
15	5	?	–	?	–	?	–
15	7	168	[1]	?	–	?	–
15	15	?	–	?	–	?	–
17	4	?	–	?	–	?	–
17	8	696	[1]	?	–	?	–
17	17	?	–	?	–	?	–
19	3	4	[7]	$11,084,874,829$	[6]	**	[6]
19	9	$7,138$	[1]	?	–	?	–
19	19	?	–	?	–	?	–

**=1,348,410,350,618,155,344,199,680,000

An integer n is called k-*admissible* if $k \leq n$ and k divides $\frac{n(n-1)}{2}$.

Chapter 2

Enumeration of Inequivalent Cycle Decompositions

In this chapter we first provide lemmas needed to establish the correctness of the backtrack algorithm we used. For the remainder of this report, fix n and k and concentrate on the enumeration of k-cycle decompositions of K_n. The vertex set will be $V = \{0, 1, \ldots, n-1\}$.

2.1 Defining cycle decomposition arrays

The following Lemma classifies all orbit representatives of the action of the symmetric group S_V on the set of cycles. First we describe some characteristics of the orbit representative of cycles.

Lemma 1 *All cycles of length k have a representation as a array of length k. Furthermore, there is a unique lexicographically smallest array of vertices, $a_0, a_1, \ldots, a_{k-1}$, each pair $\{a_i, a_{i+1}\}$* (subscript addition taken mod k) *is an edge in the cycle and*

1. $a_0 < a_i$ *for all* $1 \leq i < k$,
2. $a_1 < a_{k-1}$

Proof. By definition, there exists at least one ordering. Because the automorphism group of a k-cycle is D_{2k} there are $2k$ ways to order the vertices. Because there are rotations in D_{2k}, we can cyclically shift the array as needed for condition 1. Also because there are flips in D_{2k}, we can fix the first element and flip the rest of the cycle as needed for condition 2.

Without loss let $a_0, a_1, \ldots a_{n-1}$ be an ordering such that $a_0 < a_i$ for all $1 \leq i < k$. Because there are flips in D_{2n}, $a_0, a_{n-1}, a_{n-2}, \ldots, a_1$ is a possible ordering of the vertices.

Because these two are flips of each other, part 2 picks the lexicographically smallest one, as a_{k-1} is the reversed cycle's a_1. □

We refer to the above representation as a *cycle array*. Note that there are $\binom{n}{2}/k$ cycles in a cycle decomposition.

Lemma 2 *There are* $\binom{n}{k}\frac{(k-1)!}{2}$ *k-cycles in the complete graph.*

Proof. First we choose the vertex set. There are $\binom{n}{k}$ ways to do this. Second, we choose the ordering. By the orbit counting lemma, the number of orbits is equal to the size of the group $(k!)$ divided by the size of the automorphism group $2k$. □

Now that we have a cycle array, we can then create a unique representation for cycle decompositions.

Lemma 3 *Let* $D = \{d_1, \ldots, d_{\binom{n}{2}/k}\}$*, where* d_i*,* $1 \leq i \leq \binom{n}{2}/k$*, is a cycle, be a k-cycle decomposition of K_n. Then D has a lexicographically smallest representation as an array of vertices* $(a_0, a_1, \ldots, a_{\binom{n}{2}-1})$ *made by concatenating, in lexicographic order, the cycle arrays of each* d_i *where*

1. *Each vertex is represented* $\frac{n-1}{2}$ *times*
2. $(a_j, a_{j+1}, \ldots, a_{j+k-1})$*, where* $j = k * i$ *for some integer* $0 \leq i < \binom{n}{2}/k$*, is the ith lexicographically smallest cycle.*

Proof. Each lexicographically smallest array representing a cycle decomposition is a lexicographical ordering of all cycles contributing to the decomposition. The order is the number of edges in the graph, $\binom{n}{2}$. Note that each vertex in the complete graph has degree $(n-1)$. Also note that each vertex in a cycle has degree 2. Thus it follows that each vertex appears $\frac{n-1}{2}$ times, and consequently the first $\frac{n-1}{2}$ cycles in the array start with 0. As each cycle takes up exactly k vertices, we know that the starting position of each cycle is $0 \mod k$. The number of cycles at any position j is $\lfloor j/k \rfloor$. □

We refer to this smallest representation as a cycle decomposition array. We define two cycle decompositions to be inequivalent if their cycle decompositions arrays differ.

Alspach, Gavlas, Šajna settled that these necessary conditions are also sufficient.

Theorem 4 (Alspach-Gavlas 2001 [2], Šajna 2002 [9]) *A k-cycle decomposition of K_n exists if and only if n is k-admissible.*

We note some consequences of Theorem 4 relative to $\mathscr{M}(k, n)$. Because $\mathscr{M}(k, n)$ is the number of inequivalent cycle decompositions of K_n into k cycles, then if n is not k-admissible $\mathscr{M}(k, n) = 0$. Also if $n < k$, then $\mathscr{M}(k, n) = 0$. In this report we are concerned with computing $\mathscr{M}(k, n)$ such that n is k-admissible.

2.2 How backtracking is used.

Our overall goal is to count the number of inequivalent cycle decompositions of K_n. Our method of enumeration relies on a backtracking search. We compute $\mathscr{M}(n,k)$ by recursively attempting to extend partial solutions lexicographically.

A *partial solution* of length l to the k-cycle decomposition problem is an array $A = (a_0, a_1, \ldots, a_{l-1})$ such that the $c = \lfloor l/k \rfloor$ cycles

$$(a_{jk}, a_{jk+1}, \ldots, a_{j(k+1)-1}), j = 0, 1, \ldots, c-1$$

and the path

$$a_{ck}, a_{ck+1}, \ldots, a_{l-1}$$

are all pairwise edge disjoint. Let $E(A)$ be the set of edges covered by the partial solution A. A partial solution is a *complete solution* if $|E(A)| = \binom{n}{2}$. Let $\mathcal{A}$ be the set of all partial solutions to the k-cycle decomposition problem.

A partial solution $A = (a_0, a_1, \ldots, a_{l-1}) \in \mathcal{A}$ of length l may be extended to a partial solution $(a_0, a_1, \ldots, a_{l-1}, x) \in \mathcal{A}$ of length $l+1$ if x can be chosen such that $x \notin \{a_{ck}, a_{ck+1}, \ldots, a_{l-1}\}$ and

$$\begin{aligned} \{a_{l-1}, x\} \notin E(A) \text{ when } l \not\equiv 0, -1 \mod k \\ \{a_{l-1}, x\}, \{a_{ck}, x\} \notin E(A) \text{ when } l \equiv -1 \mod k \end{aligned}$$

where $c = \lfloor l/k \rfloor$.

In order to avoid duplication of effort in our backtrack search, we impose a total ordering "$\leq_{\text{lex}}$" on $\mathcal{A}$. If $A = (a_0, a_1, \ldots, a_{l_1-1}), B = (b_0, b_1, \ldots, b_{l_2-1}) \in \mathcal{A}$ when $a_i = b_i$ for $i = 0, 1, \ldots, j-1$, $j \leq \min\{l_1, l_2\}$, then $A \leq_{\text{lex}} B$ if $a_j < b_j$.

2.2.1 Remark on "current cycle" or "previous cycle"

In this report, when the terms *current cycle* or *new cycle* and *previous cycle* are used, there is position i indexing an array $A \in \mathcal{A}$ that is the location of this cycle. More precisely, let $j = i \mod k$, and $c = i - j$. Then current cycle and new cycle refer to the (possibly empty) subarray $a_c, \ldots, a_i$ and previous cycle refers to the (possibly empty) subarray $a_{c-k}, \ldots, a_{c-1}$. A prior cycle refers to any of the subarrays $a_{c-j*k}, \ldots, a_{c-(j-1)k-1}$ for $1 \leq j \leq \lfloor i/k \rfloor$.

2.3 Lexicographic Feasibility

In this section we describe a range of values that determine how a partial solution may be extended to a lexicographically larger partial solution.

Lemma 1 and Lemma 3 describe necessary restrictions for the "profile" of each cycle in the cycle decomposition array. By Lemma 1, the first element of each cycle must be the smallest, and the second element must be smaller than the last. By Lemma 3 each cycle must, each in turn, be lexicographically larger than the previous. For each partial solution to be extended correctly, these Lemmas are checked at every step. Lemmas 5 and 6 show that these restrictions form ranges of values $\langle s_i, e_i \rangle$.

For example, consider a partial solution $A = (5)$. One cannot extend to $A = (5, 3)$ because $3 < 5$, failing to pass the characterization of cycles set by Lemma 1 that $a_0 < a_1$.

We begin by extending a partial solution to create the first cycle.

Lemma 5 *Let $i < k$. Necessary conditions for lexicographically extending the partial solution*

$$A = (a_0, a_1, \ldots, a_{i-1})$$

to the partial solution $A = (a_0, a_1, \ldots, a_{i-1}, v)$ are that $v \in \{s_i, s_i + 1, \ldots, e_i\}$ where

$$(s_i, e_i) = \begin{cases} (0, 0) & \text{if } i = 0 \\ (1, n-2) & \text{if } i = 1 \\ (1, n-1) & \text{if } 2 \leq i \leq k-2 \\ (a_1 + 1, n-1) & \text{if } i = k-1 \end{cases}$$

Proof. By Lemma 1 the first $\frac{n-1}{2}$ cycles must begin with 0. By Lemma 1 a_0 must be the smallest in the array. By Lemma 2 $a_{k-1} > a_1$. □

In the above proof, Lemma 3 imposes no conditions because $i < k$.

The (lexicographically) largest cycle array $(n-k, n-2, n-3, \ldots, n-k-1, n-1)$ is naturally composed of the largest set of elements. By Lemma 2, the smallest element is first $a_0 = n - k$. Making a_1 the largest value then involves $a_1 < a_{k-1}$, thus $a_1 = n - 2$ and $a_{k-1} = n - 1$. The rest of the possible values are ordered in descending order. We say z is in the range $[x, y)$ to mean $x \leq z < y$ and $z \in \mathbb{Z}$.

Lemma 6 *Let $k \leq i < \binom{n}{2}$, and assume necessary conditions hold for all $0 \leq j < i$. Let $i_n = i \mod k$. Necessary conditions for lexicographically extending the partial solution*

$$A = (a_0, a_1, \ldots, a_{i-1})$$

to the partial solution $A = (a_0, a_1, \ldots, a_{i-1}, v)$ are that $v \in \{s_i, s_i + 1, \ldots, e_i\}$ where

$$(s_i, e_i) = \begin{cases} (0, 0) & \text{if } i_n = 0 \text{ and } i/k < (n-1)/2 \\ (a_{i-k}, n-k-1) & \text{if } i_n = 0 \text{ and } i/k \geq (n-1)/2 \\ (a_{i-k}, n-2) & \text{if } i_n = 1 \text{ and } a_{i-1} = a_{i-1-k} \\ (a_{i-1}, n-2) & \text{if } i_n = 1 \text{ and } a_{i-1} \neq a_{i-1-k} \\ (a_{i-i_n+1}, n-1) & \text{if } i_n = k-1 \\ (a_{i-i_n} + 1, n-1) & \text{if } i_n \notin \{0, 1, k-1\} \end{cases}$$

Proof. Note that facts mentioned in the proof for Lemma 5 now apply to the current cycle, $a_{i-i_n}, \ldots, a_{i-1}$. We split the analysis into cases based on i_n.

Let $i_n = 0$, a_i needs to be 0 for the first $\frac{n-1}{2}$ cycles. After that, the cycle needs to be the same or greater (lexicographically) than the previous cycle, so a_i in this case is always in the range $[a_{i-k}, n-k)$,where the upper bound $n-k$ comes from the fact that there needs to be $k-1$ elements in the array past this point.

Let $i_n = 1$, a_i in this case depends on the profile of the previous cycle. This new cycle needs to be lexicographically larger than the previous so if $a_{i-1} = a_{i-1-k}$ then the range is specified by $[a_{i-k}, n-1)$ where $n-1$ is not possible because then the cycle's end could not exist. Otherwise, the range is specified by $[a_{i-1}, n-1)$ because of Lemma 2.

Let $i_n = k-1$, a_i in this case depends on the first element in the cycle. The position of this vertex in the array is $i - i_n + 1$.

Let $i_n \notin \{0, 1, k-1\}$, a_i needs to be bounded by the first element in the array. □

2.4 Necessary and sufficient conditions for cycle decomposition arrays.

In this section we develop necessary and sufficient conditions for determining when an array A is a partial cycle decomposition array.

Define $I : \mathcal{A} \to \mathbb{Z}_{\binom{n}{2}}$ by $I(A)$ is the smallest position i such that one of the following conditions does not hold. Lacking this, $I(A)$ is set to the length of the array.

(I.Lexicographic Condition) With $i < \binom{n}{2}$, a_i is in the range specified by Lemma 5 and Lemma 6.

(I.Edge Condition) Let $i_n = i \mod k$ and

$$L_i = \begin{cases} \{\} & \text{if } i_n = 0 \\ \{\{a_{i-1}, a_i\}\} & \text{if } 1 \le i_n < k-1 \\ \{\{a_{i-1}, a_i\}, \{a_{i-(k-1)}, a_i\}\} & \text{if } i_n = k-1 \end{cases}.$$

Let $\mathscr{L} = \bigcup L_j$ where $0 \le j \le i-1$. Then $\mathscr{L} \cap L_i = \emptyset$.

(I.Vertex Condition) Let $i_n = i \mod k$. Then $a_i \notin \{a_{i-i_n}, \ldots, a_{i-1}\}$.

If none of the above conditions fail, we set $I(A) = |A|$.

Lemma 7 *A array D is an array of lexicographically feasible cycles if and only if conditions (I.Lexicographic Condition) and (I.Vertex Condition) of the above definition are satisfied.*

Proof. First assume D is an array of cycles. By Lemmas 5 and 6, $s_i \leq a_i \leq e_i$ and thus $I(A)$ condition (I.Lexicographic Condition) passes always. For condition (I.Vertex Condition), assume otherwise. Then there exists some j such that $I(A) = j$ as per (I.Vertex Condition). Thus there is a vertex repeated in one subarray. However, this is an array of cycles, contradiction.
Conversely, assume conditions (I.Vertex Condition) and (I.Lexicographic Condition) are never broken. Condition (I.Vertex Condition) passing directly shows that each of the arrays are cycles, as they are arrays of different vertices. Condition (I.Lexicographic Condition) passing then shows that each of these cycles pass conditions imposed by Lemma 2 and Lemma 4. □

Theorem 8 $I(A) = \binom{n}{2}$ *if and only if A is a cycle decomposition.*

Proof. First assume $I(A) = \binom{n}{2}$. By Lemma 7, $I(A) = \binom{n}{2}$ implies A is an array of cycles. By (I.Edge Condition) we know that no edge is covered twice. Then $I(A) = \binom{n}{2}$ now implies that A is a cycle decomposition.
Conversely assume A is a cycle decomposition. By Lemma 7, $I(A)$ was not decided by (I.Lexicographic Condition) or (I.Vertex Condition). A cycle decomposition implies no edges are covered twice, so thus part (I.Edge Condition) passes always. From these three passing, we know that $I(A) = \binom{n}{2}$. □

Define the set of *partial cycle decomposition arrays, PCDAs* to be K.

$$K = \{A \in \mathcal{A} : I(A) = |A|\}$$

is the set of arrays in $\mathcal{A}$ that, which upon extending with a correct vertices, could be cycle decompositions. Then our backtrack algorithm iterates through each element of K.

2.5 Algorithm Development

In this section, we present algorithm FIND-CYCLES to compute $\mathscr{M}(n,k)$. We also present a proof of correctness along with rudimentary run time analysis.

The algorithm FIND-CYCLES first finds data for the current cycle. Then it searches for new cycles, but first calculates the range of vertices it needs to iterate over by calling CALCULATE-RANGE.

At the beginning, Lines 1-6, FIND-CYCLES initializes local variables. The variable i_k refers to the position in the current cycle, c refers to the position of the start of the current cycle, $newcyc$ refers to the address of the start of the current cycle, and $prevcyc$ is a pointer to the start of the previous cycle.

Then, the basic recursive exit check on Line 7 will add one to $num\text{-}found$ if the array is large enough. Otherwise the program will continue to build arrays in Lines 8-16.

FIND-CYCLES$(g : Graph, A : int[], i : int)$

▷ k, n, and $num\text{-}found$ are all global variables.

1 $i_k \leftarrow i \pmod k$
2 $c \leftarrow i - i_k$
3 $newcyc \leftarrow \&A[i_k]$
4 $prevcyc = NULL$
5 **if** $i \geq k$
6 **then** $prevcyc \leftarrow newcyc - k$
7 **if** $i < \binom{n}{2}$
8 **then if** $i_k = 0$ and $i \neq 0$
9 **then** ADD-CYCLE-TO$(g, prevcyc)$
10 $(s_i, e_i) \leftarrow$ CALCULATE-RANGE(S, i)
11 **for** $v \leftarrow s_i$ **to** e_i
12 **do if** { g does not have the edge as specified by (I.Edge Condition) and the vertex v has not been added to this current cycle before. }
13 **then** $A[i] \leftarrow v$
14 FIND-CYCLES$(g, A, i+1)$
15 **if** $i_k = 0$ and $i \neq 0$
16 **then** REMOVE-CYCLE-FROM$(g, prevcyc)$
17 **else** $num\text{-}found \leftarrow num\text{-}found + 1$

```
CALCULATE-RANGE(A : int[], i : int)
 1  f = ⌊(i/k)⌋
 2  Switch (i_k)
 3  Case 0 :
 4          if f < (n−1)/2
 5            then s_i ← 0
 6                 ei ← 0
 7            else  s_i ← prevcyc[0]
 8                  e_i ← n − k − 1
 9          return (s_i, e_i)
10  Case 1 :
11          if i ≠ i_k
12            then if prevcyc[0] = newcyc[0]
13                   then s_i ← prevcyc[1]
14                   else  s_i ← newcyc[0] + 1
15            else  s_i ← 1
16          e_i ← n − 1
17          return (s_i, e_i)
18  Case k − 1:
19          s_i ← newcyc[1] + 1
20          e_i ← n − 1
21          return (s_i, e_i)
22  Case Default :
23          s_i ← newcyc[0] + 1
24          e_i ← n − 1
25          return (s_i, e_i)
```

First this section, Lines 8-9 will add cycles to the graph of recording, kept by pointer g. The cycles will only be added if the previous call of FIND-CYCLES finished a cycle. Similarly, the end of this section, Lines 15-16, will remove the cycle, as this call is returning and will thus replace the last element in the previous cycle.

Sandwiched between these two checks, Lines 10-14, are where $I(S)$ is insured. The main point here is that FIND-CYCLES$(g, S, i+1)$ is called after extending the array only if $I(S) = i$. The range is calculated, so that (I.Lexicographic Condition) is satisfied, then the **if** statement on line 12 checks both (I.Vertex Condition) and (I.Edge Condition).

The algorithm CALCULATE-RANGE is strictly to return the ranges as specified by Lemmas 5 and 6.

2.5.1 Backtrack search example 1: splitting K_7 into 3-cycles

As described above, FIND-CYCLES computes $\mathscr{M}(n,k)$ through attempting to extend each possible partial solution. Consequently the backtrack search exhibited below yields a proof that

$$(012)(034)(056)(135)(146)(236)(245)$$

is the lexicographically smallest 3-cycle decomposition of K_7. New lines represent new indexes evaluated. Vertices in square brackets are not possible for reasons given as a short key corresponding to the list below.

edge already taken See (I.Edge Condition). If a vertex appears within square brackets proclaiming "[1 edge already taken]" then the edge(s) this vertex would have added to the graph were already covered.

in current cycle See (I.Vertex Condition). If a vertex fails and quickly pounces to the penalty box with the exclamation "[2 in current cycle]" it means that even though this vertex is in the range specified by Lemmas 5 and 6, this vertex has already appeared in the current cycle, so the cycle would close in on itself early if this vertex were to be added.

```
( 0
( 0 1
( 0 1 2
( 0 1 2)( 0
( 0 1 2)( 0[1 edge already taken]
( 0 1 2)( 0[2 edge already taken]
( 0 1 2)( 0 3
( 0 1 2)( 0 3 4
( 0 1 2)( 0 3 4)( 0
( 0 1 2)( 0 3 4)( 0[3 edge already taken]
( 0 1 2)( 0 3 4)( 0[4 edge already taken]
( 0 1 2)( 0 3 4)( 0 5
( 0 1 2)( 0 3 4)( 0 5 6
( 0 1 2)( 0 3 4)( 0 5 6)( 0
( 0 1 2)( 0 3 4)( 0 5 6)( 0[5 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1
( 0 1 2)( 0 3 4)( 0 5 6)( 1[2 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3[4 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1[3 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4[5 edge already taken]
```

```
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 1
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 1[4 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 1[5 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3[4 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3[5 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3 6
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3 6)( 2
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3 6)( 2[3 edge already taken]
( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3 6)( 2 4
Decomp 1  ( 0 1 2)( 0 3 4)( 0 5 6)( 1 3 5)( 1 4 6)( 2 3 6)( 2 4 5)
```

2.5.2 Backtrack search example 2: splitting K_5 into 5-cycles

```
(
( 0
( 0 1
( 0 1[1 in current cycle]
( 0 1 2
( 0 1 2[1 in current cycle]
( 0 1 2[2 in current cycle]
( 0 1 2 3
( 0 1 2 3[2 in current cycle]
( 0 1 2 3[3 in current cycle]
( 0 1 2 3 4
( 0 1 2 3 4)( 0
( 0 1 2 3 4)( 0[1 edge already taken]
( 0 1 2 3 4)( 0 2
( 0 1 2 3 4)( 0 2[1 edge already taken]
( 0 1 2 3 4)( 0 2[2 in current cycle]
( 0 1 2 3 4)( 0 2[3 edge already taken]
( 0 1 2 3 4)( 0 2 4
( 0 1 2 3 4)( 0 2 4 1
Decomp 1  ( 0 1 2 3 4)( 0 2 4 1 3)
( 0 1 2 3 4)( 0 2 4 1[4 in current cycle and edge already taken]
( 0 1 2 3 4)( 0 2 4[2 in current cycle]
( 0 1 2 3 4)( 0 2 4[3 edge already taken]
( 0 1 2 3 4)( 0 2 4[4 in current cycle]
( 0 1 2 3 4)( 0 3
( 0 1 2 3 4)( 0 3 1
( 0 1 2 3 4)( 0 3 1[1 in current cycle]
```

```
( 0 1 2 3 4)( 0 3 1[2 edge already taken]
( 0 1 2 3 4)( 0 3 1[3 in current cycle]
( 0 1 2 3 4)( 0 3 1 4
( 0 1 2 3 4)( 0 3 1 4[4 in current cycle and edge already taken]
( 0 1 2 3 4)( 0 3[2 edge already taken]
( 0 1 2 3 4)( 0 3[3 in current cycle]
( 0 1 2 3 4)( 0 3[4 edge already taken]
( 0 1 2 4
( 0 1 2 4[2 in current cycle]
( 0 1 2 4 3
( 0 1 2 4 3)( 0
( 0 1 2 4 3)( 0[1 edge already taken]
( 0 1 2 4 3)( 0 2
( 0 1 2 4 3)( 0 2[1 edge already taken]
( 0 1 2 4 3)( 0 2[2 in current cycle]
( 0 1 2 4 3)( 0 2 3
( 0 1 2 4 3)( 0 2 3 1
( 0 1 2 4 3)( 0 2 3 1[3 in current cycle and edge already taken]
Decomp 2  ( 0 1 2 4 3)( 0 2 3 1 4)
```

2.5.3 Proof of correctness

In this section we prove that FIND-CYCLES$(\emptyset, a, 0)$ correctly computes $\mathscr{M}(n,k)$, where $\emptyset$ denotes the empty graph. The proof is acomplished by estsblishing the following two items on the set

$$\mathcal{I} = \{(k,n) : n \text{ is } k\text{-admissible.}\}$$

of *input parameters*.

1. Show that FIND-CYCLES halts on parameters in $\mathcal{I}$.
2. Show that FIND-CYCLES gives the correct output.

We begin with two useful observations that can be easily deduced from the algorithm.

Observation 9 *Let $A = (a_0, \ldots, a_{i-1})$ be a partial solution. Then* CALCULATE-RANGE(A, i) *returns (s_i, e_i) the start and end range as specified by Lemmas 5 and 6.*

Observation 10 *The line 12 in* FIND-CYCLES *correctly checks if a vertex v passes $I(A)$ edge check (I.Edge Condition) that v passes $I(A)$ vertex check (I.Vertex Condition).*

Lemma 11 shows that FIND-CYCLES halts for given input parameters.

Lemma 11 *Let $(k,n) \in \mathcal{I}$ and let A an array of size $\binom{n}{2}$. Then* FIND-CYCLES$(\emptyset, S, 0)$ *halts.*

Proof. Let FIND-CYCLES(X, A, i) be an arbitrary call in the call stack for the algorithm. Then line 11 insures that there is a finite branch factor, that FIND-CYCLES will be called a finite number of times.

Continuing, note that line 11 is only reached when $i < \binom{n}{2}$, ensuring a finite depth. Thus there is a finite branching factor, and a finite depth factor, and thus the algorithm always returns. □

The relation between line 14 and the function $I(A)$ is made clear in Lemma 12.

Lemma 12 *Let $j < \binom{n}{2}$ and let S be an array of size $\binom{n}{2}$. Also let $A = S_0, \ldots, S_{j-1}$ such that A is a $PCDA$ and let X be the graph defined as the union of all the cycles completed in A. Then line 14 of* FIND-CYCLES(X, S, j) *is only called forming PCDAs with length j in which $I(A_{new}) = |A_{new}| = j$.*

Proof. From Observation 9 we know that the position added will not be outside the range defined by (I.Lexicographic Condition). From Observation 10 we know that an position added will pass the checks defined by (I.Edge Condition) or (I.Vertex Condition). Thus since $j < \binom{n}{2}$, $I(A_{new}) = |A_{new}| = j$. □

Thus we have shown that FIND-CYCLES will only be called with admissible values of $I(A)$. We now show how the set traversed is well ordered.

Theorem 13 *Let S an array of size $\binom{n}{2}$. Then, at the beginning of each call to* FIND-CYCLES(X, S, i)*, each admissible partial cycle system that is lexicographically smaller than $A = S_0, \ldots, S_{i-1}$ been visited already.*

Proof. Assume otherwise, then there exists a lexicographically smaller partial cycle system array $L = (l_i)$ with an position of difference j such that $a_i = l_i$ for $i < j$ and $a_j >_{lex} l_j$ that has not been visited by FIND-CYCLES. Call this subarray of equality J. However, since A was built by FIND-CYCLES then we can conclude there was a call FIND-CYCLES(g, J, j). During this call $j < \binom{n}{2}$. From Observation 9 we know that for each admissible next vertex, FIND-CYCLES was called. However this contradicts our assumption that L was not visited. □

We conclude with the following theorem.

Theorem 14 *Let $\emptyset$ be the empty graph and S an array of size $\binom{n}{2}$. Then the algorithm* FIND-CYCLES$(\emptyset, S, 0)$ *correctly computes $\mathscr{M}(n, k)$.*

Proof. By Lemma 12, FIND-CYCLES(X, A, nc) will be called with $I(A) = \binom{n}{2}$. Thus line 17 will be called and A will be counted towards $\mathscr{M}(n, k)$. Thus every cycle decomposition array will be counted at least once. And by Theorem 13, every cycle decomposition array will be counted just once, and in lexicographic order. □

2.5.4 Note on running time

We see that our search has a branching factor with upper limit n and a depth factor of upper limit $\binom{n}{2}$. Thus we can say this algorithm is bounded above by $\binom{n}{2}^n \in O(n^{2n})$. There is only constant factor pruning done in this algorithm. Lines 8-9 and 15-16 take time $O(k)$ for every cycle added, which happens $O(n^2/k)$ times. This is completely absorbed into the $O(n^{2n})$ term.

2.6 Results and Future Projects

The two new numbers found with the algorithm developed in this report are $\mathscr{M}(9,4) = 1,643,040$ and $\mathscr{M}(9,6) = 222,243,840$. $\mathscr{M}(11,5)$ was running for 6 weeks and ended early due to a power failure.

The algorithm FIND-CYCLES could be extended and generalized by confining the modification to CALCULATE-RANGE. If one were to change this method to use the vertex degrees of the vertices in the graph covered by complete cycles and the graph passed in, then FIND-CYCLES could be used to enumerate cycle decompositions of any graph by starting with the complement of the graph in question. Also, if one were to modify CALCULATE-RANGE in such a way to give a choice set instead of a range, and one were to create a profile for arbitrary graphs, then this algorithm could be used for arbitrary graph decompositions.

In [**?**], a general algorithm is presented that gives the same labeling for every graph in an automorphism graph. *nauty*[8](No AUTomorphisms, Yes?) is a C package released by Brendan McKay to find automorphisms of graphs. It was the program [1] used to find $C(n,k)$, $k \neq 3$ in Table 1.1. In fact, because *nauty* has this canonical form (McKay's Canonical Form, MCF) algorithm built in, one could simply test edges added against the MCF of the graph required and not have to develop their own profile ordering of vertices.

One could obtain better search tree pruning by adding isomorph rejection. This can be done in two simple steps: 1. Adding a global data structure to store the graphs, a modified suffix tree is what I would suggest. 2. folding lines 8-9 and 15-16 on the inside of the for loop, and adding an isomorph against the suffix tree collecting totals of graphs that have nontrivial automorphisms, while ignoring this check if the automorphism group is trivial. Because the C code used *nauty* graphs, this is just one call to the algorithm *nauty* with the correct options set. Adding isomorphism rejection increases computation time at each search node. So if substantial pruning is not achieved then this could result in slower performance.

Bibliography

[1] P. Adams and D. E. Bryant, *Cyclically generated closed m-trail systems of order $(2m + 1)$, $m \leq 10$*, J. Combin. Math. Combin. Comput. **17** (1995), 3–19. MR 1315637 (96f:05113)

[2] B. Alspach and H. Gavlas, *Cycle decompositions of K_n and $K_n - I$*, J. Combin. Theory Ser. B **81** (2001), no. 1, 77–99. MR 1809427 (2002e:05034)

[3] D. Bryant, *Cycle decompositions of complete graphs*, Surveys in combinatorics 2007, London Math. Soc. Lecture Note Ser., vol. 346, Cambridge Univ. Press, Cambridge, 2007, pp. 67–97. MR 2252790 (2008k:05163)

[4] C. J. Colbourn, *Hamiltonian decompositions of complete graphs*, Ars Combin. **14** (1982), 261–270.

[5] Bryś K. and Lone Z., *A complete solution of the Holyer problem*, 4th Twente Workshop on Graph and Combinatorial Optimization (1995).

[6] P. Kaski and P.R. Östergård, *The Steiner triple systems of order 19*, Math. Comp. **73** (2004), 2075–2092.

[7] D. L. Kreher and T. C. Frenz, *An algorithm for enumerating distinct cyclic Steiner systems*, J. Combin. Math. Combin. Comput. **11** (1992), 23–32.

[8] B. D. McKay, nauty *user's guide (version 2.4)*, `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`.

[9] M. Šajna, *Cycle decompositions. III. Complete graphs and fixed length cycles*, J. Combin. Des. **10** (2002), no. 1, 27–78. MR 1871681 (2003e:05072)

[10] D. S. Stones, *Hamilton cycle decompositions of the complete graph*, `http://mathoverflow.net/questions/10577/hamilton-cycle-decompositions`, JUN 2011.

[11] M. Tarsi and D. Dor, *Graph decomposition if NP-complete: A complete proof of Holyer's conjecture*, Siam J. Comput. **26** (1997), 1166–1187.

Appendix: Source Code

vcnums.h
```
#define num_verts 5
#define cyc_size 5
```

straight_count.c
```
/* ci is the cycle index: the number of cycles in
the graph *after* this prog is run */
/* it describes the level of the graph we are at */

#include "vcnums.h"
#define MAXN num_verts
#include "gtools.h"
#include <limits.h>

#define GRAPHSIZET ((MAXM*MAXN*WORDSIZE)/CHAR_BIT)
#define NC2 (((-1 + num_verts) * num_verts) / 2 )
#define DEBUG_COUNT 3
#define MAX(n,m) ((n>m)? n: m)

graph tacosauce[num_verts] = {0};
static const graph empty_graph[num_verts] = {0};
long long unsigned num_found = 0;
long long unsigned rec_calls = 0;

void print_one_cycle(int *cyc)
{
  int i;
  for( i = 0; i < cyc_size; i++)
    printf("%2d ", cyc[i]);
  printf("\n");
}

int aresame(graph* g1, graph* g2)
{
  int i = memcmp(g1, g2, GRAPHSIZET);
  return !(!(!(i)));
}

int has_edge(graph *g, int i, int j)
{
  set *gv;
  gv = GRAPHROW(g, i, MAXM);
  return ISELEMENT(gv, j);
}
```

```
void remove_edge(graph *g, int i, int j)
{
  set *gv = GRAPHROW(g, i, MAXM);
  DELELEMENT(gv, j);
  gv = GRAPHROW(g, j, MAXM);
  DELELEMENT(gv, i);
}

void add_edge(graph *g, int i, int j)
{
  set *gv = GRAPHROW(g, i, MAXM);
  ADDELEMENT(gv, j);
  gv = GRAPHROW(g, j, MAXM);
  ADDELEMENT(gv, i);
}

void remove_cycle_from(graph *g, int *cyc)
{
  int i;
  for(i = 0; i < cyc_size; ++i)
    remove_edge(g, cyc[i], cyc[(i+1)%cyc_size]);
}

void add_cycle_to(graph *g, int *cyc)
{
  int i;
  for(i = 0; i < cyc_size; ++i)
    add_edge(g, cyc[i], cyc[(i+1)%cyc_size]);
}

int has_edge_check(int index, graph *g, int *cyc, int pot)
{
  switch(index)
  {
    case 0:
      return 1;
    case cyc_size-1:
      if(has_edge(g, cyc[0], pot))
        return 0;
    default:
      return !(has_edge(g, cyc[index-1], pot));
  }
}
```

```
int edge_repeat_check(int index, int *cyc, int pot)
{
  int i;
  for( i = 0; i < index; i++)
    if( cyc[i] == pot )
      return 0;
  return 1;
}

void calculate_start_end(int *si, int *ei,
    int newindex, int *prevcyc, int index,
    int *newcyc, int *cyc)
{
  int f =  (index / cyc_size);
  switch(newindex)
  {
    case 0:
      si[0] = (prevcyc)? prevcyc[0] : 0;
      if( f < (num_verts-1)/ 2 )
        ei[0] = 0+1;
      else
        ei[0] = num_verts;
      break;
    case 1:
      si[0] = (prevcyc)? ( (prevcyc[0] == newcyc[0])?
        prevcyc[1]: newcyc[0]+1) : cyc[0] + 1;
      ei[0] = num_verts-1;
      break;
    case cyc_size-1:
      si[0]= newcyc[1]+1;
      ei[0] = num_verts;
      break;
    default:
      si[0] = newcyc[0]+1;
      ei[0] = num_verts;
  }
}

#if DEBUG_COUNT >= 2
void print_reason(int index, int newindex, int* cyc,
  char* after)
{
    int i;
```

```
    printf("(");
    for(i = 0; i < index; i++)
    {
      if(!(i % cyc_size) && i)
        printf(")(");
      printf("%2d", cyc[i]);
    }
    printf("%s", after);
    printf("\n");
}
#endif
void add_cycles(graph *g, int* cyc, int index)
{

  int i,j,si,lrc,ei;
  int newindex = index % cyc_size;
  int cycstart = index - newindex;
  int *newcyc = cyc+cycstart;
  int *prevcyc = 0;
#if DEBUG_COUNT >= 2
  char after[100] = {0};
  if( rec_calls && cyc[0]!=0)
    exit(0);
#endif
  rec_calls++;
  lrc = rec_calls;
  if(newcyc != cyc)
    prevcyc = newcyc - cyc_size;
#if DEBUG_COUNT >= 3
  if(num_found>1)
    exit(0);
#endif
  if(index < NC2)
  {
    calculate_start_end(&si, &ei, newindex, prevcyc,
      index, newcyc, cyc);
#if DEBUG_COUNT >= 2
    if(ei > num_verts)
      exit(0);
    print_reason(index, newindex, cyc, "");
    for(i = 0; i < si; i++)
    {
      sprintf(after, "[%d silow]", i);
```

```
      print_reason(index, newindex, cyc, after);
    }
#endif
    if(!newindex && index)
      add_cycle_to(g, prevcyc);
    for(i=si; i<ei; ++i)
    {
      if( has_edge_check( newindex, g, newcyc, i) &&
          edge_repeat_check(newindex, newcyc, i))
      {
        cyc[index]=i;
        add_cycles(g, cyc, index+1);
      }
#if DEBUG_COUNT >= 2
      else
      {
        if( ! has_edge_check( newindex, g, newcyc, i) &&
          ! edge_repeat_check(newindex, newcyc, i) )
        {
          sprintf(after,
            "[%d in current cycle and edge already taken]", i);
          print_reason(index, newindex, cyc, after);
        }
        else if( ! edge_repeat_check(newindex, newcyc, i))
        {
          sprintf(after, "[%d in current cycle]", i);
          print_reason(index, newindex, cyc, after);
        }
        else if( ! has_edge_check( newindex, g, newcyc, i) )
        {
          sprintf(after, "[%d edge already taken]", i);
          print_reason(index, newindex, cyc, after);
        }
      }
#endif
    }
#if DEBUG_COUNT >= 2
    for(i = ei; i < num_verts; i++)
    {
      sprintf(after, "[%d eihigh]", i);
      print_reason(index, newindex, cyc, after);
    }
#endif
```

```
    if(!newindex && index)
      remove_cycle_from(g, prevcyc);
  }
  else
  {
    num_found++;
#ifdef DEBUG_COUNT
    printf("Decomp %d  ", num_found);
    sprintf(after, ")");
    print_reason(NC2,0,cyc,after);
#endif
  }

}

int main()
{
  int cyc[NC2] = {0};
  graph work[num_verts] = {0};
  //setvbuf(stdout, (char *) NULL, _IONBF, 0);

  add_cycles(work, cyc, 0);
  printf("Found %llu decompositions of K_%d into %d cycles.\n",
    num_found, num_verts, cyc_size);

#ifdef DEBUG_COUNT
  printf("add_cycles was called %llu times. \n", rec_calls);
#endif
}
```